Clase 6 — 18.12.25

#VSC #javascript #DOM

👩‍🏫 Profesora: Sara Gonzalo
📘 Desarrollo de interfaces. JAVASCRIPT, JQUERY, REALIDAD VIRTUAL
🗓️ Clase 6 — 18/12/2025
🎯 **Tema:** DOM, jerarquía de nodos y creación dinámica de elementos

---

1️⃣ ¿Qué es el DOM?

El **DOM (Document Object Model)** es la **representación interna que hace el navegador del documento HTML** en forma de **estructura arbórea** (árbol de nodos).

📌 **Idea clave:**
JavaScript **no trabaja directamente con el HTML**, trabaja con el **DOM**.

🔹 Definición sencilla

> El DOM es un árbol de objetos que representa todos los elementos HTML de una página y sus relaciones.

🔹 ¿Por qué es tan importante?

Gracias al DOM podemos:

- Crear elementos HTML desde JavaScript
- Modificar elementos existentes
- Cambiar estilos y atributos
- Añadir eventos dinámicamente
- Construir páginas **sin escribir HTML**

👉 **Objetivo final del curso**:

> Tener un HTML mínimo y generar toda la página desde JavaScript.

2️⃣ Jerarquía del DOM (estructura en árbol)

El **DOM** no es una lista plana de elementos, sino una **estructura jerárquica en forma de árbol**. Esta jerarquía es fundamental porque define **cómo se relacionan los elementos entre sí** y **cómo JavaScript puede acceder y modificarlos**.

Cuando el navegador carga un documento HTML, no lo interpreta como texto sin más, sino que lo transforma internamente en un **árbol de nodos**, donde cada elemento ocupa una posición concreta dentro de esa jerarquía. Esa posición determina quién es su padre, quiénes son sus hijos y en qué contexto existe ese elemento.

Podemos imaginarlo como un árbol genealógico: ningún elemento existe de forma aislada, todos dependen de otro.

A nivel conceptual, la jerarquía básica del DOM parte siempre de este esquema:

```
document
 └── html
     ├── head
     └── body
         ├── div
         │   └── p
         └── img
```

Este dibujo representa exactamente cómo el navegador entiende la página.

🔹 Nodo raíz

El **nodo raíz** del DOM es siempre el objeto:

```
document
```

Esto es muy importante, porque aunque en HTML veamos que la etiqueta principal es `<html>`, desde el punto de vista de JavaScript **el verdadero punto de entrada es `document`**.

`document` representa **todo el documento HTML completo**:

- contiene el `<html>`
- contiene el `<head>`
- contiene el `<body>`
- y contiene cualquier nodo que se cree dinámicamente después

Por eso, cuando trabajamos con el DOM, siempre partimos de `document`. No existe ningún nodo "por encima" de él.

🔹 Nodos hijos y relaciones jerárquicas

Dentro de esta estructura, los elementos se relacionan entre sí mediante **relaciones padre → hijo**.

Un **nodo hijo** es cualquier elemento que está contenido dentro de otro. Por ejemplo:

- `<html>` es hijo directo de `document`
- `<head>` y `<body>` son hijos de `<html>`
- un `<div>` puede ser hijo de `<body>`
- un `<p>` puede ser hijo de un `<div>`

Estas relaciones **no son opcionales** ni conceptuales:
son **obligatorias** para que el DOM funcione.

📌 En el DOM **no existen nodos "sueltos"**.
Todo nodo debe colgar de otro nodo.

Esta idea es clave para entender por qué, cuando creamos elementos desde JavaScript, **no aparecen en la página hasta que los añadimos a un padre**.

3️⃣ Concepto clave: nodos y relaciones

En el DOM, absolutamente **todo es un nodo**. No solo las etiquetas HTML visibles, sino también:

- el texto que hay dentro de una etiqueta
- las imágenes
- los atributos
- incluso los comentarios del HTML

Sin embargo, en esta fase del curso el foco está puesto principalmente en dos tipos:

- **Nodos de tipo elemento**, como `div`, `p`, `img`, `span`
- **Nodos de texto**, que contienen el contenido textual

Esto es importante porque, cuando manipulamos el DOM, JavaScript **no distingue entre "HTML bonito" y "HTML feo"**, solo trabaja con nodos y relaciones.

🔹 Regla fundamental del DOM

> Un nodo **no existe en el DOM** hasta que se añade a un padre.

Este punto suele generar confusión, así que conviene dejarlo muy claro:

- Crear un nodo **no significa** que exista en la página
- Un nodo puede existir solo en memoria
- Solo pasa a formar parte del DOM cuando se inserta en la jerarquía

Por eso se dice:

> **Crear ≠ insertar**

Esta distinción es clave para entender por qué el navegador no muestra nada hasta que usamos `appendChild`.

4️⃣ Flujo de trabajo con nodos (muy importante)

La profesora insiste mucho en que **antes de escribir código**, hay que pensar en la estructura. El trabajo con el DOM no es improvisado: sigue un **flujo mental muy concreto**.

Primero debemos pensar **qué nodo queremos crear**. No es lo mismo un `div` que un `span` o una `img`. Cada uno tiene un propósito distinto.

Después hay que decidir **quién será su padre directo**. Este paso es crítico, porque define:

- dónde aparecerá el elemento
- qué relación tendrá con el resto del documento
- cómo se verá en la jerarquía DOM

Solo entonces se crea el nodo con `createElement`, se guarda en una variable y, finalmente, se inserta en el DOM usando `appendChild`.

Si este último paso no se ejecuta, el nodo **no aparece en la página**, porque nunca ha pasado a formar parte del árbol del DOM.

📌 Esta es una de las ideas más importantes de toda la clase:

> **Hasta que no se ejecuta `appendChild`, el nodo no pertenece al DOM.**

5️⃣ Crear nodos con `createElement`

Crear nodos con JavaScript es el primer paso para **construir HTML de forma dinámica**. El método que se utiliza para ello es `createElement`, que permite generar cualquier etiqueta HTML directamente desde código JavaScript.

Cuando ejecutamos una instrucción como:

```
let mispan = document.createElement("span");
```

JavaScript crea un **nuevo nodo de tipo elemento**, en este caso un `<span>`. Sin embargo, este nodo **no pertenece todavía al documento HTML**. Existe únicamente en memoria, dentro del motor de JavaScript, y por tanto **no se muestra en pantalla**.

Es muy importante entender este punto, porque visualmente no ocurre nada. El navegador no puede renderizar un nodo que todavía no forma parte del árbol DOM. En este momento:

- el nodo existe como objeto JavaScript
- se puede guardar en una variable
- se pueden modificar sus propiedades
- pero **no está integrado en la página**

Esto refuerza una idea fundamental del DOM: **crear un nodo no implica que exista en el documento**.

🔹 Error típico al empezar con el DOM

Un error muy habitual cuando se empieza a trabajar con `createElement` es pensar que, por el simple hecho de crear el nodo, este va a aparecer automáticamente en la página.

Esto no ocurre porque el DOM funciona de forma jerárquica. Hasta que un nodo **no tiene un padre**, el navegador no lo considera parte del documento. Por tanto, el nodo no se renderiza ni se tiene en cuenta visualmente.

Este error es muy común y suele provocar la sensación de que “el JavaScript no funciona”, cuando en realidad el nodo nunca ha sido insertado en el DOM.

6️⃣ El padre del nodo: concepto clave

Antes de insertar cualquier nodo, es imprescindible tener claro **dónde va a vivir dentro de la jerarquía del DOM**. Es decir, debemos decidir **quién será su padre directo**.

El DOM no permite nodos aislados. Cada nodo debe colgar de otro. Por ejemplo:

- un `span` suele estar contenido dentro de un `div`
- un nodo de texto suele estar contenido dentro de un `p`
- un `p` suele estar contenido dentro del `body`

Esta decisión no es un detalle menor: define la estructura final del documento y afecta directamente a cómo se renderiza el contenido y cómo se comportan los estilos y los eventos.

📌 En el DOM, la relación padre → hijo **es obligatoria**.
Si no se define un padre, el nodo no puede formar parte del documento.

7️⃣ Insertar nodos en el DOM (`appendChild`)

Una vez creado el nodo y decidido cuál será su padre, el siguiente paso es **insertarlo en la jerarquía del DOM**. Esto se hace mediante el método `appendChild`.

Por ejemplo:

```
document.appendChild(mispan);
```

Con esta instrucción estamos diciendo explícitamente que el nodo `mispan` pasa a ser **hijo directo de `document`**. A partir de este momento:

- el nodo ya forma parte del DOM
- aparece en el HTML generado por el navegador
- se renderiza en pantalla
- puede recibir estilos y eventos

Este es el momento exacto en el que el nodo “cobra vida” dentro de la página.

⚠️ Aclaración importante sobre `appendChild`

El método `appendChild` tiene un comportamiento muy concreto:

- añade el nodo **como hijo** del elemento padre
- lo coloca **siempre al final** de la lista de hijos de ese padre

Esto significa que, si un elemento ya tiene otros hijos, el nuevo nodo se insertará después de ellos. Por tanto, el orden en el que se usan los `appendChild` **sí importa** y determina la estructura final del DOM.

📌 **Idea clave para cerrar este bloque**:

> En el trabajo con el DOM, crear nodos es solo el principio.
> Lo realmente importante es **insertarlos correctamente en la jerarquía**, respetando siempre la relación padre → hijo.

8️⃣ Ejemplo completo: jerarquía controlada

Una vez comprendido que el DOM funciona como una estructura jerárquica, es fundamental ver **cómo se construye esa jerarquía de forma consciente desde JavaScript**. Este ejemplo es especialmente importante porque refleja exactamente el tipo de razonamiento que se espera a partir de ahora en clase.

El objetivo no es solo crear elementos, sino **decidir explícitamente dónde van a vivir dentro del DOM**.

Queremos generar desde JavaScript la siguiente estructura HTML:

```
<div>
  <p></p>
</div>
```

Aquí no hay nada complejo a nivel visual, pero sí mucho contenido conceptual. Tenemos dos elementos y una relación clara: el `<p>` está contenido dentro del `<div>`. Esa relación debe reproducirse exactamente en el DOM cuando trabajamos desde JavaScript.

El primer paso consiste en **crear los nodos**, uno a uno:

```
let midiv = document.createElement("div");
let parrafo1 = document.createElement("p");
```

En este punto, ambos nodos existen únicamente en memoria. JavaScript ya sabe que `midiv` es un `div` y que `parrafo1` es un `p`, pero **ninguno de los dos pertenece todavía al documento**. No están en pantalla ni forman parte del DOM.

El siguiente paso es **definir la jerarquía**, es decir, decidir quién cuelga de quién. Este paso es el que da sentido a todo el proceso.

```
document.appendChild(midiv);
midiv.appendChild(parrafo1);
```

Estas dos líneas son mucho más importantes de lo que parecen. La primera indica que el `div` pasa a ser hijo directo del documento. La segunda indica que el `p` pasa a ser hijo directo del `div`. Con esto, el navegador ya puede construir internamente la estructura correcta del DOM.

Traducido a lenguaje conceptual:

- el `div` se inserta en el documento
- el `p` se inserta dentro del `div`

A partir de ese momento, el navegador entiende la página como una estructura jerárquica real, no como elementos sueltos.

Si lo visualizamos mentalmente, el resultado es exactamente este:

```
document
 └── div (midiv)
     └── p (parrafo1)
```

Este ejercicio no busca “hacer algo bonito”, sino **entrenar la forma correcta de pensar el DOM**: primero estructura, después contenido y comportamiento.

9️⃣ Por qué este enfoque es tan importante

Este modo de trabajar con el DOM es clave porque cambia por completo la forma de entender una página web. En lugar de pensar el HTML como algo fijo, empezamos a verlo como una estructura **dinámica y manipulable desde JavaScript**.

Gracias a este enfoque, se obtiene un control total del contenido de la página desde el código. Se pueden crear interfaces que cambian en función de datos, acciones del usuario o eventos del navegador, sin necesidad de escribir HTML adicional.

Además, esta forma de trabajar es la base de prácticamente todos los frameworks modernos. Aunque más adelante se utilicen herramientas que abstraen este proceso, internamente todas siguen exactamente el mismo principio: creación de nodos, definición de jerarquías y actualización del DOM.

Por último, entender este proceso elimina la sensación de que el DOM es algo “mágico” o impredecible. Cuando se domina la jerarquía y el flujo de creación, el comportamiento del navegador deja de ser confuso y pasa a ser completamente lógico.

📌 Sin esta base, el DOM parece magia negra.
📌 Con esta base, el DOM es simplemente una estructura bien organizada.

🔟 Acceso a los nodos del DOM

Una vez que entendemos cómo se construye la jerarquía del DOM, el siguiente paso natural es **aprender a acceder a los nodos que ya existen** dentro de esa estructura.

Acceder a un nodo significa poder referirnos a él desde JavaScript para leer información, modificar su contenido, cambiar estilos, añadir eventos o reutilizarlo como contenedor para otros nodos. Sin este acceso, el DOM sería una estructura pasiva, imposible de manipular.

Aquí aparece una idea fundamental que se repetirá constantemente a partir de ahora:

> Si no puedes seleccionar un nodo, no puedes trabajar con él.

Todo el trabajo con el DOM —ya sea crear nodos, modificarlos o reaccionar a eventos— empieza siempre por **localizar el nodo correcto dentro de la jerarquía**. A partir de este punto entran en juego los métodos de selección (`getElementById` , `querySelector` , etc.), que permiten “apuntar” a elementos concretos del árbol DOM y empezar a interactuar con ellos.

Este paso conecta directamente con lo visto en la clase anterior y prepara el terreno para manipular la página de forma cada vez más avanzada.

1️⃣1️⃣ El objeto `document`

En JavaScript, todo el trabajo con el DOM comienza siempre desde un único punto de entrada: el objeto global `document` .

```
document
```

Este objeto representa **el documento HTML completo** que el navegador ha cargado y procesado. No es una etiqueta concreta ni un fragmento del HTML, sino la **representación global del documento entero**. Desde `document` se puede acceder a absolutamente todo lo que forma parte de la página: elementos, texto, atributos, eventos y la estructura jerárquica del DOM.

Cuando se dice que “JavaScript controla el DOM”, en realidad lo que se quiere decir es que **JavaScript controla el objeto `document`** , y a través de él puede interactuar con todos los nodos que cuelgan de esa estructura.

Por eso, cualquier operación relacionada con:

- seleccionar elementos
- crear nodos
- añadir eventos
- modificar estilos
- reaccionar a la carga de la página

empieza siempre usando `document` como punto de partida.

📌 En términos conceptuales:

> `document` es la puerta de entrada al DOM.

1️⃣2️⃣ Selección por ID — `getElementById()`

Una vez entendido que todo parte de `document`, el siguiente paso lógico es **localizar nodos concretos dentro de la jerarquía**. El método más sencillo y directo para hacerlo es `getElementById`.

Este método permite seleccionar **un único elemento** a partir del valor de su atributo `id`.

```
document.getElementById("foto");
```

Aquí es importante remarcar que el atributo `id` debe ser **único en todo el documento HTML**. Esta unicidad es precisamente lo que permite que JavaScript pueda localizar un elemento de forma rápida y sin ambigüedades.

🧪 Ejemplo en HTML

```
<img id="foto" src="imagen.jpg">
```

En este caso, el navegador crea un nodo correspondiente a esa imagen y lo registra dentro del DOM con el identificador `foto`.

🧪 Ejemplo en JavaScript

```
let imagen = document.getElementById("foto");
```

O alternativamente:

```
const imagen = document.getElementById("foto");
```

Ambas opciones son correctas. La diferencia entre usar `let` o `const` no afecta al funcionamiento del DOM, sino a cómo gestionamos la variable en JavaScript:

- `let` se utiliza cuando prevemos que la referencia puede cambiar
- `const` se recomienda cuando la referencia al nodo va a mantenerse estable

En la práctica, cuando seleccionamos un nodo del DOM, lo habitual es usar `const`, ya que normalmente no se cambia qué nodo estamos apuntando, sino que se modifican sus propiedades.

🧠 Qué obtenemos realmente

Aunque estemos seleccionando una imagen, el valor que se guarda en la variable **no es una imagen “normal”**, sino un **objeto nodo del DOM**. Ese objeto contiene:

- propiedades (por ejemplo, `src`, `style`, `id`)
- métodos
- la información necesaria para interactuar con ese elemento dentro del documento

Esto es clave para entender por qué luego podemos hacer cosas como cambiar estilos, añadir eventos o modificar atributos directamente desde JavaScript.

1️⃣3️⃣ Selección por clase — `getElementsByClassName()`

Mientras que `getElementById` está pensado para seleccionar **un único elemento**, el método `getElementsByClassName` permite seleccionar **varios elementos a la vez** que comparten una misma clase.

```
document.getElementsByClassName("clase");
```

Este método no devuelve un solo nodo, sino una **colección de nodos**. Es decir, un conjunto de elementos que cumplen la condición indicada.

🧪 Ejemplo en HTML

```
<p class="texto">Uno</p>
<p class="texto">Dos</p>
```

Aquí el navegador crea dos nodos `<p>` distintos, ambos con la clase `texto`.

🧪 Selección desde JavaScript

```
let parrafos = document.getElementsByClassName("texto");
```

En este punto, la variable `parrafos` **no contiene un solo elemento**, sino una lista de nodos. Esta lista se comporta de forma similar a un array, aunque técnicamente es una colección del DOM.

Por eso, no se puede aplicar directamente una propiedad a `parrafos`, sino que es necesario acceder a uno de sus elementos concretos.

🔹 Acceso a un elemento concreto de la colección

```
parrafos[0].style.color = "red";
```

Con esta instrucción estamos:

- accediendo al primer elemento de la colección
- modificando su estilo
- actuando sobre un nodo concreto del DOM

Este detalle es importante porque refuerza una idea clave: **cuando seleccionamos varios nodos, debemos decidir explícitamente sobre cuál queremos actuar**.

📌 **Idea clave para cerrar este bloque**:

> Seleccionar nodos es imprescindible para trabajar con el DOM.
> Sin una referencia clara a un nodo, JavaScript no puede modificar nada.

1️⃣4️⃣ `querySelector()` — selección moderna

Con la evolución del lenguaje y de la forma de trabajar con el DOM, JavaScript incorporó métodos de selección más flexibles y potentes. El más importante de ellos es `querySelector`, que permite

seleccionar elementos del DOM utilizando **selectores CSS**, exactamente igual que se hace en una hoja de estilos.

Cuando utilizamos:

```
document.querySelector("p");
```

JavaScript busca en el DOM el **primer elemento** que coincida con ese selector. Si existen varios elementos que cumplen la condición, solo se devuelve el primero que aparece en el documento, siguiendo el orden del DOM.

Este método es especialmente útil porque **no se limita a un tipo concreto de selección**. Puede trabajar con etiquetas, clases, identificadores y combinaciones complejas, lo que lo convierte en una herramienta muy versátil.

Cómo funciona realmente `querySelector`

Internamente, el navegador recorre el árbol del DOM y aplica el selector CSS indicado. En cuanto encuentra una coincidencia válida, detiene la búsqueda y devuelve ese nodo como resultado.

Esto significa que:

- no es necesario saber de antemano si el elemento tiene `id` o clase
- no hay que cambiar de método según el tipo de selector
- el código resulta más limpio y coherente

🧪 Ejemplos de uso

```
document.querySelector("p");
```

Selecciona el **primer** `<p>` que encuentre en el documento.

```
document.querySelector(".texto");
```

Selecciona el **primer elemento** que tenga la clase `texto`.

```
document.querySelector("#foto");
```

Selecciona el elemento con `id="foto"`.

En todos los casos, el resultado es siempre **un único nodo del DOM** o `null` si no existe ningún elemento que coincida con el selector.

🧠 Por qué es una gran ventaja

La gran ventaja de `querySelector` es que **unifica todos los métodos de selección en uno solo**. No es necesario memorizar distintos métodos para id, clase o etiqueta. Basta con conocer la sintaxis de los selectores CSS, que ya se utiliza en otros contextos del desarrollo web.

Por este motivo, `querySelector` se ha convertido en el método preferido en muchos proyectos modernos, ya que reduce la complejidad del código y mejora su legibilidad.

1️⃣5️⃣ `querySelectorAll()` — selección múltiple

Cuando necesitamos seleccionar **más de un elemento** utilizando selectores CSS, se utiliza el método `querySelectorAll`.

```
document.querySelectorAll("p");
```

Este método devuelve **todos los elementos** del DOM que coincidan con el selector indicado. A diferencia de `querySelector`, aquí no se detiene en la primera coincidencia, sino que recorre todo el árbol del DOM y recoge todas las coincidencias.

El resultado no es un único nodo, sino una **NodeList**, es decir, una colección de nodos.

🧪 Ejemplo práctico

```
let parrafos = document.querySelectorAll("p");

parrafos.forEach(p => {
  p.style.color = "blue";
});
```

En este ejemplo, JavaScript selecciona todos los párrafos del documento y recorre la colección uno a uno. Para cada nodo, se modifica su estilo, cambiando el color del texto.

Este patrón es muy habitual en el trabajo con el DOM, ya que permite aplicar cambios de forma masiva sin duplicar código.

Diferencia conceptual importante

Aunque `querySelectorAll` devuelve algo similar a un array, en realidad se trata de una **NodeList**. Esto significa que:

- se puede recorrer con `forEach`
- no es un array puro
- su contenido refleja los nodos seleccionados en el momento de la llamada

Para el nivel de esta asignatura, basta con entender que es una lista de nodos sobre la que se puede iterar.

1️⃣6️⃣ Comparación entre métodos de selección

Existen varias formas de seleccionar elementos del DOM, pero no todas se usan con la misma frecuencia en la práctica profesional.

- `getElementById` es muy rápido y claro cuando se trabaja con identificadores únicos.
- `getElementsByClassName` permite seleccionar varios elementos, pero devuelve una colección menos flexible.
- `querySelector` permite seleccionar un único elemento usando cualquier selector CSS.
- `querySelectorAll` permite seleccionar múltiples elementos con total flexibilidad.

Por este motivo, en proyectos modernos se tiende a priorizar el uso de `querySelector` y `querySelectorAll`, ya que ofrecen un enfoque unificado y más expresivo para trabajar con el DOM.

📌 **Idea clave para cerrar este bloque**:

> Cuanto mejor se dominan los métodos de selección, más fácil resulta manipular el DOM de forma limpia y eficiente.

1️⃣7️⃣ Manipulación de nodos existentes

Una vez que hemos aprendido a **seleccionar un nodo del DOM**, ese nodo deja de ser algo pasivo. A partir de ese momento, JavaScript puede **modificarlo directamente**, cambiando su aspecto, su contenido o su comportamiento. Esta capacidad es una de las claves que hacen que el DOM sea dinámico.

Manipular un nodo existente significa trabajar sobre un elemento que **ya forma parte de la jerarquía del DOM**, ya sea porque estaba definido en el HTML original o porque se ha creado dinámicamente desde JavaScript.

🎨 Modificación de estilos desde JavaScript

Una de las manipulaciones más habituales es el cambio de estilos. Cada nodo del DOM dispone de una propiedad `style` que permite modificar directamente las reglas CSS de ese elemento.

Por ejemplo:

```
imagen.style.background = "red";
imagen.style.width = "200px";
imagen.style.height = "200px";
```

En este caso, JavaScript está actuando sobre el mismo nodo que el navegador usa para renderizar la página. El cambio es inmediato y visible, sin necesidad de recargar el documento.

Es importante entender que:

- estos estilos se aplican **en línea** (inline)
- tienen prioridad sobre muchas reglas CSS externas
- afectan solo al elemento seleccionado

Este mecanismo es muy potente y se utiliza constantemente para crear efectos visuales, animaciones simples o cambios de estado (por ejemplo, al pasar el ratón por encima de un elemento).

✏️ Modificación del contenido del nodo

Además de estilos, también se puede cambiar el **contenido** de un nodo. Para ello se utiliza, entre otros, la propiedad `textContent`.

```
parrafo.textContent = "Texto cambiado";
```

Con esta instrucción, el texto que había dentro del elemento se sustituye completamente por el nuevo contenido indicado. JavaScript no añade texto, sino que **reemplaza el contenido existente**.

Este tipo de manipulación es muy común cuando:

- se muestran resultados

- se actualiza información
- se responde a una acción del usuario

📌 En este punto ya se ve claramente que el DOM no es estático: los nodos pueden cambiar a lo largo del tiempo según lo que haga el usuario o el propio programa.

1️⃣8️⃣ Conexión con lo visto en clase

Todo lo trabajado hasta ahora no son conceptos aislados. La selección y manipulación de nodos existentes prepara directamente el terreno para un paso más avanzado: **la creación dinámica de nodos**.

Hasta este momento:

- hemos aprendido a localizar elementos dentro del DOM
- hemos visto cómo modificar su aspecto y contenido
- hemos entendido que todo pasa por el objeto `document`

Este conocimiento es imprescindible para poder crear nuevos nodos, insertarlos en la jerarquía correcta y, posteriormente, aplicarles estilos y eventos.

👉 Justo este es el salto que se empieza a dar en el siguiente bloque.

1️⃣9️⃣ Crear nodos dinámicamente

Hasta ahora el trabajo se centraba en **elementos que ya existían** en el HTML. A partir de aquí, el enfoque cambia: ya no dependemos del HTML escrito previamente, sino que **podemos generar nuevos elementos directamente desde JavaScript**.

Este cambio es muy importante porque implica que:

- el HTML deja de ser algo fijo
- la estructura de la página puede construirse en tiempo de ejecución
- la interfaz puede adaptarse dinámicamente a datos y acciones

📌 La idea clave es clara:

> Crear nodos nos permite construir HTML sin escribir HTML.

🔹 El método base: `createElement()`

El método que permite crear nuevos elementos es `createElement`.

```
document.createElement("span");
```

Con esta instrucción, JavaScript crea un nuevo nodo de tipo elemento, en este caso un `<span>`. Sin embargo, este nodo:

- no tiene contenido
- no tiene estilos
- no pertenece todavía al DOM
- no se muestra en pantalla

Es decir, el nodo existe, pero **solo en memoria**.

🧪 Ejemplo real y su significado

```
let mispan = document.createElement("span");
```

Aquí ocurre algo muy concreto: JavaScript crea un objeto que representa un nodo `<span>` y lo guarda en la variable `mispan`. A partir de este momento:

- podemos modificar sus propiedades
- podemos añadirle estilos
- podemos asignarle atributos
- pero el navegador aún no lo renderiza

Esto refuerza una idea fundamental que se repetirá constantemente en el trabajo con el DOM:

> Crear un nodo no es lo mismo que insertarlo en el documento.

El siguiente paso será precisamente **insertar ese nodo en la jerarquía del DOM**, definiendo quién será su padre y haciendo que el navegador lo tenga en cuenta.

2️⃣0️⃣ Regla de oro: crear ≠ mostrar

Este es uno de los conceptos **más importantes de toda la clase** y una de las causas más habituales de errores cuando se empieza a trabajar con el DOM.

Cuando se ejecuta una instrucción como:

```
document.createElement("span");
```

JavaScript crea correctamente el nodo, pero **no ocurre nada visible en la página**. Esto no significa que el código esté mal, sino que el nodo **no se ha insertado todavía en la jerarquía del DOM**. El navegador solo renderiza aquellos nodos que forman parte del árbol DOM; todo lo demás existe únicamente en memoria.

Por eso, este planteamiento es incorrecto:

```
document.createElement("span"); // no aparece
```

Aquí el nodo se crea, pero se pierde inmediatamente porque:

- no se guarda en ninguna variable
- no se inserta en ningún padre
- nunca pasa a formar parte del documento

La forma correcta de trabajar sigue siempre un flujo claro y ordenado. Primero se crea el nodo, después se guarda en una variable para poder manipularlo, y finalmente se añade a un nodo padre mediante `appendChild`. Solo en ese último paso el navegador puede renderizarlo.

Esta distinción entre **crear** y **mostrar** es fundamental para entender el funcionamiento del DOM y evita muchos problemas de “no se ve nada en pantalla”.

2️⃣1️⃣ Insertar nodos en la jerarquía DOM

Una vez creado el nodo y almacenado en una variable, el siguiente paso es **insertarlo en la jerarquía del DOM**. El método principal para hacerlo es `appendChild`.

```
padre.appendChild(hijo);
```

Este método indica de forma explícita que un nodo pasa a ser **hijo directo de otro nodo**. A partir de ese momento, el navegador lo considera parte del documento y lo incluye en el proceso de renderizado.

La traducción conceptual de esta instrucción es muy clara:

> "Añade este nodo como hijo del padre".

🧪 Ejemplo básico

```
let mispan = document.createElement("span");
document.appendChild(mispan);
```

En este ejemplo:

- primero se crea el nodo `<span>`
- después se añade directamente como hijo del objeto `document`

Como resultado, el `span`:

- pasa a formar parte del DOM
- se incluye en el HTML generado por el navegador
- se renderiza en pantalla (aunque no tenga contenido visible)

Este es el momento exacto en el que el nodo deja de ser "invisible" y se convierte en un elemento real de la página.

2️⃣2️⃣ Pensar siempre en el nodo padre

Antes de utilizar `appendChild`, es imprescindible detenerse a pensar **cuál debe ser el padre directo del nodo que estamos creando**. Esta decisión define completamente la estructura final del documento HTML.

Por ejemplo:

- un `span` suele tener como padre un `div`
- un `p` suele colgar de un `div` o del `body`
- una `img` puede colgar directamente del `body` o de un contenedor

Esta elección no es arbitraria. Determina:

- la posición del elemento en la página
- su relación con otros nodos
- cómo se aplicarán los estilos
- cómo se comportarán los eventos

📌 En el DOM, la relación padre → hijo **define la estructura final del HTML**.
Si se elige mal el padre, la página puede no comportarse como se espera, aunque el código sea técnicamente correcto.

📌 **Idea clave para cerrar este bloque**:

> El DOM no se construye “porque sí”.
> Se construye **pensando la jerarquía antes de escribir el código**.

2️⃣3️⃣ Ejemplo completo: crear estructura anidada

Este ejemplo es especialmente importante porque reúne **varios conceptos clave del DOM en un solo flujo lógico**: creación de nodos, jerarquía padre-hijo e inserción en el documento. No es un ejemplo “técnico” sin más, sino un ejercicio de **pensar la estructura antes de escribir código**.

El objetivo es crear desde JavaScript la siguiente estructura HTML:

```
<div>
  <p></p>
</div>
```

Aunque visualmente sea muy simple, a nivel del DOM implica una relación jerárquica clara: el párrafo está contenido dentro del `div`, y el `div` está contenido dentro del documento.

El primer paso consiste en **crear los nodos**, pero todavía sin insertarlos:

```
let midiv = document.createElement("div");
let parrafo1 = document.createElement("p");
```

En este punto, ambos nodos existen únicamente como objetos JavaScript. El navegador sabe qué tipo de nodos son, pero todavía **no forman parte del DOM**, no tienen padre y no se renderizan.

El siguiente paso es **insertarlos respetando el orden jerárquico**:

```
document.appendChild(midiv);
midiv.appendChild(parrafo1);
```

Estas dos líneas son las que realmente construyen la estructura. Primero se añade el `div` al documento, convirtiéndolo en hijo directo de `document`. Después, se añade el `p` como hijo directo del `div`. El orden no es casual: si intentáramos añadir el párrafo antes de que el `div` estuviera en el DOM, la jerarquía no tendría sentido.

Desde el punto de vista semántico, el navegador interpreta esto así:

- el `div` cuelga del documento
- el `p` cuelga del `div`

📌 En este momento queda definida de forma explícita la jerarquía del DOM. El navegador ya puede renderizar esa estructura y tratarla como parte real de la página.

Este ejemplo entrena una habilidad clave: **pensar el DOM como una estructura de relaciones**, no como una sucesión de instrucciones sueltas.

2️⃣4️⃣ Modificar nodos creados desde JavaScript

Una vez que un nodo ha sido creado y almacenado en una variable, **se comporta exactamente igual que cualquier nodo seleccionado desde el HTML**. No hay ninguna diferencia entre un elemento

“original” y uno creado dinámicamente.

Esto significa que podemos modificarlo libremente: cambiar estilos, atributos, contenido o añadir eventos.

Un ejemplo típico es la aplicación de estilos directamente desde JavaScript:

```
let mispan1 = document.createElement("span");

mispan1.style.background = "red";
mispan1.style.width = "200px";
mispan1.style.height = "200px";
```

Aquí JavaScript está definiendo propiedades visuales del nodo antes incluso de insertarlo en el DOM. Estas propiedades se almacenan en el objeto nodo y se aplicarán automáticamente cuando el elemento sea renderizado.

Desde el punto de vista conceptual, esto equivale a escribir CSS, pero de forma dinámica y controlada desde el código.

🧠 Observación importante sobre CSS y JavaScript

Cuando se trabaja con estilos desde JavaScript, hay que tener en cuenta una diferencia sintáctica importante. Las propiedades CSS que en una hoja de estilos se escriben con guiones, en JavaScript se escriben en **camelCase**.

Por ejemplo:

- `background-color` en CSS
- `backgroundColor` en JavaScript

```
mispan1.style.backgroundColor = "red";
```

Este detalle es puramente sintáctico, pero es una fuente frecuente de errores cuando se empieza a manipular estilos desde JS.

2️⃣5️⃣ Estilos antes o después del `appendChild`

Una duda habitual es si los estilos deben aplicarse antes o después de insertar el nodo en el DOM. La respuesta es que **ambas opciones son válidas**.

```
// aplicar estilos antes
mispan.style.color = "white";
document.appendChild(mispan);

// aplicar estilos después
document.appendChild(mispan);
mispan.style.color = "white";
```

En ambos casos, el resultado visual es exactamente el mismo. Esto ocurre porque:

- los estilos se almacenan en el objeto nodo
- el navegador los aplica cuando renderiza el elemento
- el momento exacto en el que se asignan no altera el resultado final

La elección entre una opción u otra suele depender de la claridad del código o del flujo lógico que se quiera seguir, no de una diferencia técnica.

📌 **Idea clave para cerrar este bloque**:

> Un nodo creado desde JavaScript es un ciudadano de primera clase del DOM.
> Puede estilizarse, modificarse y comportarse exactamente igual que cualquier elemento HTML.

2️⃣6️⃣ Modificar atributos HTML — `setAttribute()`

Hasta ahora hemos visto cómo modificar **estilos** de un nodo usando la propiedad `style`, pero los elementos HTML tienen muchos más atributos además del aspecto visual. Para trabajar con ellos, JavaScript proporciona el método `setAttribute`.

Este método permite **añadir o modificar cualquier atributo HTML** de un elemento, independientemente de si ese atributo afecta a la presentación, a la funcionalidad o a la identificación del nodo.

La sintaxis es muy directa:

```
elemento.setAttribute("atributo", "valor");
```

Conceptualmente, esta instrucción equivale a escribir un atributo directamente en el HTML, pero lo hace de forma dinámica y controlada desde JavaScript.

🧪 Ejemplo con imágenes

Un caso muy habitual es la creación dinámica de imágenes. Partimos de un documento HTML que **no contiene ninguna etiqueta** `<img>`, y es JavaScript quien se encarga de crearla y configurarla.

```
let imagen1 = document.createElement("img");
document.appendChild(imagen1);

imagen1.setAttribute("src", "fondo1.jpg");
```

Aquí ocurren varias cosas importantes:

- primero se crea el nodo `<img>`
- después se inserta en el DOM
- finalmente se le asigna el atributo `src`

Como resultado, el navegador interpreta exactamente lo mismo que si el HTML contuviera esta línea:

```
<img src="fondo1.jpg">
```

La diferencia es que ahora **el elemento no existe hasta que JavaScript lo crea**. Esto demuestra que el DOM puede construirse completamente en tiempo de ejecución.

🔹 Uso de otros atributos comunes

El método `setAttribute` no se limita al atributo `src`. Puede utilizarse para cualquier atributo HTML habitual.

Por ejemplo:

```
imagen1.setAttribute("width", "50%");
imagen1.setAttribute("height", "30%");
imagen1.setAttribute("id", "mifoto2");
```

Con estas instrucciones:

- se define el tamaño de la imagen
- se asigna un identificador único
- se prepara el nodo para futuras selecciones o eventos

Este último punto es especialmente importante, ya que asignar un `id` a un nodo creado dinámicamente permite **volver a acceder a él más adelante** mediante métodos de selección como `getElementById` o `querySelector`.

2️⃣7️⃣ Crear HTML completo desde JavaScript

Llegados a este punto, ya no estamos modificando pequeños detalles del HTML existente. Estamos en condiciones de **crear elementos completos desde JavaScript**, sin necesidad de que aparezcan previamente en el archivo HTML.

El siguiente ejemplo, sacado directamente de tus apuntes, ilustra perfectamente este concepto:

```
let imagen2 = document.createElement("img");
document.appendChild(imagen2);

imagen2.setAttribute("src", "foto1.jpg");
imagen2.setAttribute("width", "50%");
imagen2.setAttribute("height", "30%");
imagen2.setAttribute("id", "mifoto2");
```

Este fragmento de código genera un elemento `<img>` totalmente funcional:

- no existe ninguna etiqueta `<img>` en el HTML original
- el elemento nace desde JavaScript
- sus atributos se configuran dinámicamente
- el DOM se construye en tiempo real

Desde el punto de vista del navegador, no hay ninguna diferencia entre este elemento y uno escrito a mano en el HTML. Ambos forman parte del DOM y se comportan exactamente igual.

Este enfoque marca un cambio muy importante en la forma de trabajar: el HTML deja de ser la única fuente de estructura y JavaScript pasa a tener un papel protagonista en la construcción de la interfaz.

2️⃣8️⃣ El evento principal: `DOMContentLoaded`

Cuando se trabaja con creación de nodos, inserción en el DOM y asignación de eventos, el **momento de ejecución del código** es crítico. No basta con escribir bien el código; hay que asegurarse de que se ejecute cuando el navegador esté preparado.

Por eso, en este contexto, el evento `DOMContentLoaded` es fundamental.

Este evento se dispara cuando:

- el navegador ha cargado el HTML
- el DOM ha sido completamente construido
- la jerarquía de nodos ya existe

Si el código que crea nodos o añade eventos se ejecuta antes de este momento, pueden producirse errores porque el DOM aún no está disponible.

🧪 Estructura correcta de trabajo

```
document.addEventListener("DOMContentLoaded", function () {

  // aquí todo el código DOM
});
```

Este patrón garantiza que:

- el DOM existe
- los nodos pueden crearse e insertarse con seguridad
- los eventos pueden asociarse correctamente

📌 Por eso la profesora insiste en que **todo lo que manipule el DOM debe ir dentro de este listener**. No es una manía, es una necesidad técnica.

📌 **Idea clave para cerrar este bloque**:

> Crear y modificar nodos no depende solo del código, sino también del momento en el que se ejecuta.

Con esto queda completamente cerrada la lógica de la **Clase 6**:
DOM → nodos → jerarquía → creación → atributos → eventos → control del tiempo de ejecución.

2️⃣9️⃣ DOM + eventos: el paso definitivo

Hasta este punto de la clase, el trabajo con el DOM ha sido progresivo y muy estructurado. Primero se han creado nodos, después se han insertado en la jerarquía del documento y, a continuación, se han modificado sus estilos y atributos. Todo esto permite construir la estructura visual de una página, pero **todavía falta lo más importante**: la interacción.

El verdadero salto conceptual se produce cuando empezamos a **aplicar eventos a elementos creados desde JavaScript**. En ese momento, el DOM deja de ser una estructura estática y pasa a convertirse en un sistema dinámico que responde a acciones del usuario.

Este paso es clave porque demuestra que **no hay ninguna diferencia real** entre un elemento escrito en HTML y uno creado dinámicamente desde JavaScript. Ambos pueden:

- recibir eventos
- reaccionar al usuario
- modificar su comportamiento
- cambiar su aspecto en tiempo real

A partir de aquí, JavaScript no solo construye la página, sino que **controla completamente la experiencia del usuario**.

3️⃣0️⃣ Evento principal: `DOMContentLoaded`

Para que todo lo anterior funcione correctamente, hay una condición imprescindible: el código debe ejecutarse **cuando el DOM ya está disponible**.

Cuando trabajamos con creación de nodos y asignación de eventos, no basta con que el archivo JavaScript esté bien escrito. Es fundamental que el navegador ya haya:

- cargado el HTML
- construido la jerarquía del DOM
- preparado el documento para ser manipulado

Por este motivo, todo el código que:

- crea nodos
- añade eventos
- modifica el DOM

debe ejecutarse dentro del evento `DOMContentLoaded`.

```
document.addEventListener("DOMContentLoaded", function () {

  // todo el código DOM va aquí

});
```

Este patrón garantiza que JavaScript trabaja **en el momento correcto**, evitando errores típicos como intentar acceder a nodos que todavía no existen. No es una recomendación opcional, sino una regla técnica fundamental cuando se manipula el DOM de forma intensiva.

3️⃣1️⃣ Ejemplo base: imagen dinámica + eventos

Este ejemplo resume perfectamente **el flujo completo de trabajo con el DOM**, integrando todo lo visto hasta ahora. El objetivo es sencillo, pero muy representativo: crear una imagen desde JavaScript, insertarla en el documento y dejarla preparada para reaccionar a eventos.

En primer lugar, todo el código se encapsula dentro del evento `DOMContentLoaded`, asegurando que el DOM está listo antes de empezar a trabajar.

```
document.addEventListener("DOMContentLoaded", function () {

  let imagen2 = document.createElement("img");
  document.appendChild(imagen2);

  imagen2.setAttribute("src", "foto1.jpg");
  imagen2.setAttribute("width", "50%");
  imagen2.setAttribute("height", "30%");
  imagen2.setAttribute("id", "mifoto2");

});
```

En este fragmento ocurre un proceso completo y coherente:

- se crea un nodo `<img>` desde JavaScript
- se inserta en la jerarquía del DOM
- se configuran sus atributos
- se le asigna un identificador único

El resultado es un elemento que **no existe en el HTML original**, pero que pasa a formar parte del documento como si hubiera estado allí desde el principio. Gracias al `id`, este nodo queda preparado para ser seleccionado más adelante y recibir eventos como `mouseover`, `mouseout` o `click`.

Este ejemplo demuestra de forma clara que:

- el HTML puede ser mínimo o incluso inexistente
- JavaScript puede construir la interfaz completa
- los eventos pueden aplicarse sin ninguna limitación

A partir de este punto, ya no hay diferencia conceptual entre “HTML escrito” y “HTML generado”. Todo es DOM, y todo puede ser controlado desde JavaScript.

📌 **Idea clave para cerrar este bloque**:

> Cuando se combinan DOM + creación dinámica + eventos, JavaScript pasa a ser el verdadero motor de la página.

3️⃣2️⃣ Añadir eventos con `addEventListener`

Una vez que un nodo ha sido creado y añadido correctamente a la jerarquía del DOM, pasa a comportarse exactamente igual que cualquier elemento definido originalmente en el HTML. Esto implica que **puede recibir eventos sin ninguna limitación**.

Aquí es donde se produce el paso definitivo: **conectar la estructura creada dinámicamente con la interacción del usuario**. El método que permite hacer esto es `addEventListener`.

La condición imprescindible es que el nodo **ya exista en el DOM**. Si se intenta añadir un evento a un nodo que todavía no ha sido insertado, el navegador no podrá encontrarlo y el código fallará.

🧪 Evento `mouseover`

El evento `mouseover` se dispara cuando el ratón entra en el área del elemento. Es muy utilizado para crear efectos visuales sencillos que indican al usuario que un elemento es interactivo.

```
document.getElementById("mifoto2")
  .addEventListener("mouseover", function () {

    imagen2.style.opacity = "0.6";

});
```

En este fragmento ocurre lo siguiente:

- se selecciona el nodo mediante su `id`
- se añade un listener para el evento `mouseover`

- cuando el evento se produce, se modifica el estilo del elemento

El efecto visual es inmediato: al pasar el ratón sobre la imagen, su opacidad disminuye. Esto no es solo un efecto estético, sino una forma de **comunicación visual con el usuario**, indicando que el elemento responde a su acción.

🧪 Evento `mouseout`

El evento complementario a `mouseover` es `mouseout`, que se dispara cuando el ratón sale del área del elemento. Se utiliza normalmente para **restaurar el estado original** del elemento.

```
document.getElementById("mifoto2")
  .addEventListener("mouseout", function () {

    imagen2.style.opacity = "1";

});
```

En este caso, cuando el ratón deja de estar sobre la imagen, la opacidad vuelve a su valor original. De esta forma se completa el efecto visual, creando una interacción coherente y natural.

Este patrón (`mouseover` + `mouseout`) es muy común en interfaces web y demuestra claramente cómo los eventos permiten modificar el DOM en tiempo real.

3️⃣3️⃣ Código completo integrado

Este fragmento reúne **todo lo trabajado en la clase** en un solo ejemplo continuo y funcional. No hay nada nuevo aquí: simplemente se conectan correctamente todos los conceptos vistos.

```
document.addEventListener("DOMContentLoaded", function () {

  let imagen2 = document.createElement("img");
  document.appendChild(imagen2);

  imagen2.setAttribute("src", "foto1.jpg");
  imagen2.setAttribute("width", "50%");
  imagen2.setAttribute("height", "30%");
  imagen2.setAttribute("id", "mifoto2");

  document.getElementById("mifoto2")
    .addEventListener("mouseover", function () {
      imagen2.style.opacity = "0.6";
    });

  document.getElementById("mifoto2")
    .addEventListener("mouseout", function () {
      imagen2.style.opacity = "1";
    });

});
```

Este código sigue un flujo muy claro y correcto:

Primero, espera a que el DOM esté completamente cargado.
Después, crea un nodo dinámicamente y lo inserta en el documento.
A continuación, configura sus atributos para que sea funcional y accesible.
Por último, añade eventos que modifican su comportamiento visual en respuesta a acciones del usuario.

Desde el punto de vista del navegador, esta imagen es indistinguible de una imagen escrita directamente en HTML. La única diferencia es **quién la ha creado**: en este caso, JavaScript.

3️⃣4️⃣ Qué conceptos se trabajan aquí (visión de examen)

Este ejemplo final integra y demuestra la comprensión de varios conceptos fundamentales que son perfectamente evaluables:

Se trabaja la **creación dinámica de nodos**, entendiendo que los elementos pueden nacer desde JavaScript.
Se aplica correctamente la **inserción en la jerarquía del DOM**, respetando la relación padre–hijo.
Se utiliza la **selección por ID** para acceder a nodos concretos.
Se gestionan **eventos del ratón**, entendiendo cuándo se disparan y por qué.
Se modifican **estilos en tiempo real**, demostrando que el DOM es dinámico.
Y todo ello se encapsula dentro del **listener principal** `DOMContentLoaded`, garantizando el momento correcto de ejecución.

Este bloque no es solo un ejemplo práctico, sino un resumen completo del enfoque que se seguirá a partir de ahora en la asignatura.

📌 **Idea final para cerrar la Clase 6**:

> Cuando se dominan nodos, jerarquía, atributos y eventos, JavaScript deja de “acompañar” al HTML y pasa a **dirigir completamente la página**.

3️⃣5️⃣ Error típico muy importante

Uno de los errores más frecuentes —y más frustrantes— al trabajar con el DOM y eventos es **intentar añadir un evento a un nodo que todavía no existe** en la jerarquía del documento.

Por ejemplo:

```
// ERROR
document.getElementById("mifoto2").addEventListener(...);
```

Este código falla no porque la sintaxis sea incorrecta, sino porque **JavaScript no encuentra ningún nodo con ese `id` en el DOM en ese momento**. Desde el punto de vista del navegador, ese elemento simplemente no existe aún.

Este error suele aparecer cuando:

- el nodo se crea dinámicamente más tarde
- el código se ejecuta antes de que el DOM esté listo
- se asume que crear un nodo equivale a que ya exista en el documento

Para que un nodo pueda recibir eventos, debe cumplirse un orden muy concreto y obligatorio. Primero el nodo debe crearse, después debe añadirse a la jerarquía del DOM y **solo entonces** puede seleccionarse

y recibir eventos.

No es una cuestión de estilo ni de buenas prácticas: es una **regla técnica del funcionamiento del DOM**.

📌 Pensarlo de forma lógica ayuda mucho:
no se puede escuchar eventos de algo que todavía no existe.

3️⃣6️⃣ Concepto clave de la clase

> **El DOM es dinámico, y los eventos también lo son.**

Esta frase resume perfectamente la idea central de toda la clase. El DOM no es una estructura fija que se define una vez y ya no cambia. Al contrario, es una estructura viva que puede modificarse constantemente durante la ejecución de la página.

Desde el punto de vista del navegador, **no importa de dónde venga un elemento**:

- puede estar escrito directamente en el HTML
- puede haberse creado desde JavaScript
- puede haberse generado en respuesta a una acción del usuario

Lo único que importa es una cosa:
si el elemento **existe en el DOM**, entonces:

- puede seleccionarse
- puede modificarse
- puede recibir eventos

Esta idea rompe la falsa separación entre “HTML” y “JavaScript”. En realidad, todo acaba siendo DOM, y todo se gestiona de la misma forma.

3️⃣7️⃣ Resumen final

En esta última parte se consolidan los conceptos más importantes de toda la clase. Se demuestra que `addEventListener` funciona sin problemas sobre nodos creados dinámicamente, siempre que se respete el orden correcto de creación e inserción.

Los eventos del ratón, como `mouseover` y `mouseout`, permiten crear efectos visuales sencillos pero muy ilustrativos, que muestran claramente cómo el DOM puede reaccionar en tiempo real a la interacción del usuario.

También queda claro que el **orden de ejecución es crítico**. No basta con saber escribir el código; hay que ejecutarlo en el momento adecuado. Por eso `DOMContentLoaded` se convierte en el evento principal cuando se trabaja con manipulación intensiva del DOM.

Este bloque no añade nuevos conceptos aislados, sino que **une todo lo visto**: nodos, jerarquía, atributos, eventos y control del tiempo de ejecución.

📌 **Idea final para cerrar la Clase 6**:

> Entender el DOM no es aprender métodos de memoria, sino comprender **cuándo existen los nodos, cómo se relacionan y en qué momento pueden reaccionar**.

🧩 Cierre de la Clase 6

Con esta clase ya sabes:

- cómo funciona el DOM
- cómo crear HTML desde JS
- cómo aplicar estilos
- cómo añadir eventos dinámicos

👉 Estás a un paso de:

> **Construir páginas completas solo con JavaScript.**

3️⃣8️⃣ Práctica entregable: Formulario de notas con `addEventListener` + DOM

Este ejercicio integra directamente lo visto en la Clase 6: **DOM (jerarquía), selección de nodos, creación/uso de nodos existentes, y eventos con** `addEventListener`, ejecutándolo todo en el momento correcto con `DOMContentLoaded`.

La idea es que el **HTML solo define la estructura**, y **JavaScript controla la lógica**, enlazando el comportamiento con eventos (submit/click), tal y como se explicó en clase.

🌳 Árbol del proyecto

```
practica-notas/
├─ index.html
├─ css/
│  └─ styles.css
├─ js/
│  └─ app.js
└─ img/
   ├─ aprobado.png
   ├─ suspenso.png
   ├─ notable.png
   ├─ sobresaliente.png
   └─ matricula.png
```

📌 Si no tienes imágenes todavía, no pasa nada: la práctica funciona igual.
(La parte de imagen es "decorativa" y demuestra `setAttribute()`.)

✅ Código completo — index.html

> Observa que enlazo el JS con `defer`. Esto cumple el objetivo de clase: **asegurar que el DOM está listo antes de tocar nodos**, sin depender de colocar el `<script>` al final del `<body>`.

```html
<!doctype html>
<html lang="es">
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width,initial-scale=1" />
```

```
  <title>Práctica addEventListener — Notas</title>

  <link rel="stylesheet" href="css/styles.css" />

  <!-- defer = el navegador descarga el JS pero lo ejecuta cuando el HTML ya está
parseado -->
  <script src="js/app.js" defer></script>
</head>

<body>
  <header class="header">
    <h1>Práctica <span class="pill">addEventListener</span></h1>
    <p class="sub">Formulario de notas + cálculo de media + calificación + alertas</p>
  </header>

  <main class="container">
    <section class="card">
      <h2>📋 Datos del alumno</h2>

      <!-- Formulario: evento principal será submit -->
      <form id="formNotas" autocomplete="off" novalidate>
        <div class="grid2">
          <div class="field">
            <label for="nombre">Nombre</label>
            <input id="nombre" name="nombre" type="text" placeholder="Ej: David"
required />
            <small class="hint">Obligatorio</small>
          </div>

          <div class="field">
            <label for="apellido1">Apellido 1</label>
            <input id="apellido1" name="apellido1" type="text" placeholder="Ej:
Rodríguez" required />
            <small class="hint">Obligatorio</small>
          </div>

          <div class="field">
            <label for="apellido2">Apellido 2</label>
            <input id="apellido2" name="apellido2" type="text" placeholder="Ej: Igual"
required />
            <small class="hint">Obligatorio</small>
          </div>
        </div>

        <h3 class="mt">🧱 Notas</h3>
        <div class="grid3">
          <div class="field">
            <label for="n1">Asignatura 1</label>
            <input id="n1" name="n1" type="number" min="0" max="10" step="0.1"
placeholder="0 - 10" required />
            <small class="hint">0 a 10 (admite decimales)</small>
          </div>

          <div class="field">
```

```
            <label for="n2">Asignatura 2</label>
            <input id="n2" name="n2" type="number" min="0" max="10" step="0.1"
placeholder="0 - 10" required />
            <small class="hint">0 a 10 (admite decimales)</small>
          </div>

          <div class="field">
            <label for="n3">Asignatura 3</label>
            <input id="n3" name="n3" type="number" min="0" max="10" step="0.1"
placeholder="0 - 10" required />
            <small class="hint">0 a 10 (admite decimales)</small>
          </div>
        </div>

        <div class="actions">
          <button id="btnCalcular" type="submit">Calcular</button>
          <button id="btnLimpiar" type="button" class="secondary">Limpiar</button>
        </div>
      </form>
    </section>

    <section class="card">
      <h2>📌 Resultado</h2>

      <!-- Zona DOM dinámica: el JS reemplaza su contenido (efecto de sustitución) -->
      <div id="resultado" class="result">
        <p class="muted">Rellena el formulario y pulsa <b>Calcular</b>.</p>
      </div>

      <figure class="badge">
        <img id="imgCalif" src="" alt="Imagen de calificación" />
        <figcaption id="textoCalif" class="muted">Sin calificación
todavía</figcaption>
      </figure>

      <div class="legend">
        <h3>📚 Rangos de calificación</h3>
        <ul>
          <li><b>SUSPENSO</b>: 0 – 4,9</li>
          <li><b>APROBADO</b>: 5 – 6,9</li>
          <li><b>NOTABLE</b>: 7 – 8,9</li>
          <li><b>SOBRESALIENTE</b>: 9 – 9,9</li>
          <li><b>MATRÍCULA DE HONOR</b>: 10</li>
        </ul>
      </div>
    </section>
  </main>

  <footer class="footer">
    <p>Objetivo: controlar DOM y eventos desde JavaScript.</p>
  </footer>
</body>
</html>
```

✅ Código completo — css/styles.css

```css
:root{
  --bg:#0b1220;
  --card:#121a2a;
  --text:#e8eefc;
  --muted:#a9b6d3;
  --line: rgba(255,255,255,.10);
  --accent:#4ea1ff;
}

*{ box-sizing:border-box; }
body{
  margin:0;
  font-family: system-ui, -apple-system, Segoe UI, Roboto, Arial, sans-serif;
  color:var(--text);
  background: radial-gradient(1200px 600px at 20% 0%, rgba(78,161,255,.25),
transparent 60%),
              radial-gradient(900px 500px at 90% 20%, rgba(172,78,255,.18),
transparent 55%),
              var(--bg);
}

.header{
  padding: 28px 18px 10px;
  text-align:center;
}
.header h1{
  margin:0;
  font-size: clamp(22px, 3vw, 34px);
}
.sub{
  margin:8px 0 0;
  color:var(--muted);
}
.pill{
  background: rgba(78,161,255,.18);
  border:1px solid rgba(78,161,255,.35);
  padding: 3px 10px;
  border-radius: 999px;
  font-size: .9em;
}

.container{
  width:min(1100px, 92vw);
  margin: 18px auto 40px;
  display:grid;
  grid-template-columns: 1fr;
  gap: 14px;
}
@media (min-width: 980px){
  .container{ grid-template-columns: 1.2fr .8fr; }
}
```

```
.card{
  background: rgba(18,26,42,.85);
  border: 1px solid var(--line);
  border-radius: 16px;
  padding: 16px;
  box-shadow: 0 10px 30px rgba(0,0,0,.25);
}

h2{ margin: 0 0 12px; }
h3{ margin: 12px 0 8px; color: var(--muted); font-weight: 600; }
.mt{ margin-top: 14px; }

.grid2{
  display:grid;
  grid-template-columns: 1fr;
  gap: 10px;
}
.grid3{
  display:grid;
  grid-template-columns: 1fr;
  gap: 10px;
}
@media (min-width: 640px){
  .grid2{ grid-template-columns: 1fr 1fr; }
  .grid3{ grid-template-columns: 1fr 1fr 1fr; }
}

.field label{
  display:block;
  margin-bottom:6px;
  font-weight: 600;
}
.field input{
  width:100%;
  padding: 10px 12px;
  border-radius: 12px;
  border:1px solid var(--line);
  background: rgba(255,255,255,.05);
  color: var(--text);
  outline: none;
}
.field input:focus{
  border-color: rgba(78,161,255,.6);
  box-shadow: 0 0 0 4px rgba(78,161,255,.12);
}
.hint{
  display:block;
  margin-top:6px;
  color: var(--muted);
}

.actions{
  display:flex;
```

```
  gap:10px;
  margin-top: 14px;
  flex-wrap: wrap;
}
button{
  border:0;
  padding: 10px 14px;
  border-radius: 12px;
  cursor:pointer;
  font-weight: 700;
}
button#btnCalcular{
  background: var(--accent);
  color: #08111f;
}
button.secondary{
  background: rgba(255,255,255,.08);
  color: var(--text);
  border: 1px solid var(--line);
}

.result{
  padding: 12px;
  border-radius: 14px;
  border:1px dashed rgba(255,255,255,.18);
  background: rgba(255,255,255,.03);
}
.muted{ color: var(--muted); }

.badge{
  margin: 12px 0 0;
  padding: 12px;
  border-radius: 14px;
  border: 1px solid var(--line);
  display:flex;
  align-items:center;
  gap: 12px;
}
.badge img{
  width: 64px;
  height: 64px;
  border-radius: 12px;
  object-fit: cover;
  border:1px solid rgba(255,255,255,.12);
  background: rgba(255,255,255,.06);
}

.legend ul{
  margin: 8px 0 0;
  padding-left: 18px;
  color: var(--muted);
}

.footer{
```

```
  text-align:center;
  color: var(--muted);
  padding: 12px 10px 22px;
}
```

✅ Código completo — js/app.js (comentado en profundidad y ligado a los apuntes)

```
"use strict";

/**
 * ✅ CLASE 6 (DOM + NODOS + EVENTOS)
 *
 * Esta práctica une TODO lo visto:
 * - DOM = estructura jerárquica (nodos padre/hijo)
 * - Selección de nodos: getElementById / querySelector
 * - Eventos con addEventListener (submit, click)
 * - Evento principal DOMContentLoaded (ejecución cuando el DOM está listo)
 * - Modificación dinámica del DOM:
 *    - “Efecto de sustitución” (cambiar contenido de un nodo)
 *    - setAttribute para atributos HTML
 *    - style para estilos CSS desde JS
 */

document.addEventListener("DOMContentLoaded", function () {
  /**
   * 🧠 1) DOMContentLoaded (evento principal)
   * En clase: “El evento principal es DOMContentLoaded”.
   *
   * Razón:
   * - Si intentas seleccionar elementos antes de que existan en el DOM
   *   (por ejemplo con getElementById), obtendrás null.
   * - Es el error típico: añadir eventos a un nodo que aún no existe.
   *
   * Con DOMContentLoaded te aseguras de que:
   * - el navegador ya ha construido la jerarquía DOM
   * - ya puedes seleccionar nodos y asociarles eventos
   */

  // ============================================================
  // 🧩 2) Acceso a nodos (selección) -> getElementById()
  // ============================================================
  // En tus apuntes: “Si no puedes seleccionar un nodo, no puedes trabajar con él.”
  // Aquí seleccionamos los nodos que ya existen en el HTML (nodos existentes),
  // para luego manipularlos desde JS.

  const form = document.getElementById("formNotas");
  const btnLimpiar = document.getElementById("btnLimpiar");

  const nombre = document.getElementById("nombre");
  const apellido1 = document.getElementById("apellido1");
  const apellido2 = document.getElementById("apellido2");
```

```
  const n1 = document.getElementById("n1");
  const n2 = document.getElementById("n2");
  const n3 = document.getElementById("n3");

  // Nodo donde pintamos el resultado -> aquí aplicaremos el “efecto de sustitución”
  const resultado = document.getElementById("resultado");

  // Nodo imagen + texto asociados al resultado (setAttribute y textContent)
  const imgCalif = document.getElementById("imgCalif");
  const textoCalif = document.getElementById("textoCalif");

  // ============================================================
  // 🎯 3) Eventos: comportamiento = acción (evento) + función
  // ============================================================

  /**
   * ✅ Evento submit del formulario
   * En vez de usar onclick en HTML, lo hacemos desde JS:
   * - queda todo centralizado en un único sitio (JS)
   * - se evita mezclar lógica (JS) con estructura (HTML)
   * - se sigue el objetivo final: “controlar la página desde JS”
   */
  form.addEventListener("submit", function (e) {
    /**
     * ⛔ Importante:
     * Un formulario al hacer submit recarga la página por defecto.
     * Eso rompería nuestra práctica porque perderíamos:
     * - el resultado pintado
     * - el control del DOM
     *
     * Por eso evitamos el comportamiento por defecto:
     */
    e.preventDefault();

    // ============================================================
    // 🧾 4) Leer valores del formulario (DOM -> JS)
    // ============================================================
    // value = leer lo que el usuario ha escrito.
    // trim() = elimina espacios al inicio y final (evita "   " como nombre válido).
    const alumno = `${nombre.value.trim()} ${apellido1.value.trim()}
${apellido2.value.trim()}`.trim();

    // Validación mínima de datos personales
    if (!nombre.value.trim() || !apellido1.value.trim() || !apellido2.value.trim()) {
      alert("Faltan datos del alumno (nombre y apellidos).");
      return; // cortamos la función (no seguimos)
    }

    // ============================================================
    // 🔢 5) Convertir notas a números y validar rango
    // ============================================================
    // En HTML recibimos strings; para calcular necesitamos números reales.
    const notas = [
```

```
      parseFloat(n1.value),
      parseFloat(n2.value),
      parseFloat(n3.value),
    ];

    // Si alguna no es número -> NaN
    if (notas.some((x) => Number.isNaN(x))) {
      alert("Introduce las 3 notas (números válidos).");
      return;
    }

    // Rango 0..10
    if (notas.some((x) => x < 0 || x > 10)) {
      alert("Las notas deben estar entre 0 y 10.");
      return;
    }

    // ============================================================
    // 🧮 6) Cálculo de media
    // ============================================================
    // Media aritmética simple:
    const media = (notas[0] + notas[1] + notas[2]) / 3;

    // ============================================================
    // 🏷️ 7) Calificación según enunciado (función reutilizable)
    // ============================================================
    const calif = obtenerCalificacion(media);

    // ============================================================
    // 🚦 8) Reglas de promoción (alertas)
    // ============================================================
    // - Si 1 nota < 5: asignatura pendiente, no prácticas
    // - Si 2 o más < 5: repetir curso
    // - Si todas >= 5: puede hacer prácticas
    const suspensos = notas.filter((x) => x < 5).length;

    if (suspensos >= 2) {
      alert("Tienes 2 o más asignaturas suspendidas: debes repetir el curso.");
    } else if (suspensos === 1) {
      alert("Tienes una asignatura pendiente: no puedes iniciar prácticas hasta
aprobarla.");
    } else {
      alert("Todas las asignaturas aprobadas: puedes iniciar tu período de
prácticas.");
    }

    // ============================================================
    // 🖥️ 9) Pintar resultado en el DOM (efecto de sustitución)
    // ============================================================
    pintarResultado({ alumno, notas, media, calif });
  });

  /**
   * ✅ Evento click: limpiar
```

```
   * Igual que en clase: “podemos añadir listener a cualquier elemento”.
   * Aquí lo añadimos al botón.
   */
  btnLimpiar.addEventListener("click", function () {
    // reset() devuelve el formulario a estado inicial
    form.reset();

    // Sustituimos el contenido del nodo resultado (efecto sustitución)
    resultado.innerHTML = `<p class="muted">Rellena el formulario y pulsa
<b>Calcular</b>.</p>`;

    // Quitamos la imagen y devolvemos el texto
    imgCalif.removeAttribute("src");
    textoCalif.textContent = "Sin calificación todavía";
  });

  // ============================================================
  // 🧩 10) Funciones auxiliares (reutilización de código)
  // ============================================================

  /**
   * Determina la calificación en función de la media.
   * Respetamos literalmente los rangos del enunciado.
   */
  function obtenerCalificacion(media) {
    // Matrícula solo si es EXACTAMENTE 10
    if (media === 10) return "MATRÍCULA DE HONOR";

    // 9 - 9,9 (según enunciado)
    if (media >= 9 && media <= 9.9) return "SOBRESALIENTE";

    // 7 - 8,9
    if (media >= 7 && media <= 8.9) return "NOTABLE";

    // 5 - 6,9
    if (media >= 5 && media <= 6.9) return "APROBADO";

    // 0 - 4,9
    return "SUSPENSO";
  }

  /**
   * Pinta los resultados en pantalla.
   * Aquí aplicamos:
   * - innerHTML (sustitución)
   * - setAttribute para src / alt
   * - textContent para texto limpio
   */
  function pintarResultado({ alumno, notas, media, calif }) {
    const mediaTxt = media.toFixed(2);

    // 🔁 Sustituimos el contenido del div resultado
    resultado.innerHTML = `
      <p><b>Alumno:</b> ${escapeHTML(alumno)}</p>
```

```
    <p><b>Notas:</b> ${notas.map((x) => x.toFixed(1)).join(" · ")}</p>
    <p><b>Media:</b> ${mediaTxt}</p>
    <p><b>Calificación:</b> <span class="pill">${calif}</span></p>
  `;

  // Imagen según calificación (atributos HTML)
  const ruta = imagenPorCalificacion(calif);

  if (ruta) {
    // setAttribute() -> modificar atributos HTML (apuntes)
    imgCalif.setAttribute("src", ruta);
    imgCalif.setAttribute("alt", `Imagen: ${calif}`);
  } else {
    imgCalif.removeAttribute("src");
  }

  // Texto de apoyo
  textoCalif.textContent = `Calificación final: ${calif}`;
}

/**
 * Devuelve la ruta de imagen en función de la calificación.
 * Demuestra una práctica real: asociar "estado" -> "UI/imagen".
 */
function imagenPorCalificacion(calif) {
  switch (calif) {
    case "SUSPENSO":
      return "img/suspenso.png";
    case "APROBADO":
      return "img/aprobado.png";
    case "NOTABLE":
      return "img/notable.png";
    case "SOBRESALIENTE":
      return "img/sobresaliente.png";
    case "MATRÍCULA DE HONOR":
      return "img/matricula.png";
    default:
      return "";
  }
}

/**
 * Seguridad/limpieza:
 * Evita que el alumno pueda meter HTML dentro del resultado.
 * (No es obligatorio para clase, pero es buena práctica y no molesta.)
 */
function escapeHTML(str) {
  return str
    .replaceAll("&", "&amp;")
    .replaceAll("<", "&lt;")
    .replaceAll(">", "&gt;")
    .replaceAll('"', "&quot;")
    .replaceAll("'", "&#039;");
```

```
  }
});
```

✅ Conexión directa con los apuntes (lo que estás demostrando en esta práctica)

- **Nodo raíz**: `document` como entrada a todo.
- **Selección por ID**: `getElementById()` para acceder a nodos existentes del HTML.
- **Eventos con listener**: `addEventListener("submit")`, `addEventListener("click")`.
- **Evento principal**: `DOMContentLoaded` para evitar el error típico de "nodo aún no existe".
- **Efecto de sustitución**: `resultado.innerHTML = ...` para pintar salida dinámica.
- **Atributos HTML desde JS**: `setAttribute("src", ...)` para la imagen temática.
- **Objetivo final**: lógica y comportamiento centralizados en JS.

3️⃣9️⃣ Por qué `submit` es mejor que `onclick` en formularios

Cuando trabajamos con un **formulario**, el evento natural no es el `click` del botón, sino el **`submit`** del propio `<form>`.

🔹 Qué ocurre si usas `onclick`

- Funciona "a simple vista" si el usuario pulsa el botón con el ratón.
- Pero **no cubre** otros comportamientos típicos de formularios:
  - Pulsar **Enter** dentro de un input para enviar.
  - Accesibilidad (teclado y lectores).

📌 Si solo usas `onclick`, el usuario puede "enviar" sin pasar por tu lógica.

🔹 Ventajas reales de `submit`

✅ Captura *todas* las formas de envío del formulario.
✅ Centraliza la lógica: el formulario "manda" y JS decide qué hacer.
✅ Te permite bloquear el comportamiento por defecto (recarga) con:

```
e.preventDefault();
```

🧠 Enlace directo con Clase 6 (DOM + eventos)

> **Evento = acción + función**
> En formularios, la acción importante es *enviar datos*, así que el evento lógico es `submit`.

4️⃣0️⃣ `innerHTML` vs `textContent` y el "efecto de sustitución"

En clase se menciona el **efecto de sustitución**: cuando tú "pintas" contenido desde JS, normalmente estás **reemplazando lo anterior**.

🔹 `innerHTML` (sustitución con HTML)

Sirve cuando quieres escribir **estructura HTML** (etiquetas).

Ejemplo (como en la práctica):

```
resultado.innerHTML = `
  <p><b>Alumno:</b> ${alumno}</p>
  <p><b>Media:</b> ${media.toFixed(2)}</p>
`;
```

✅ Permite `<b>`, `<p>`, `<ul>`, etc.
⚠️ Si metes texto del usuario directamente, puede ser peligroso si no lo limpias (por eso añadimos `escapeHTML()` como buena práctica).

🔹 `textContent` (sustitución solo texto)

Sirve cuando solo quieres cambiar el contenido sin HTML:

```
textoCalif.textContent = `Calificación final: ${calif}`;
```

✅ Más seguro (no interpreta etiquetas).
✅ Ideal para textos sueltos.

🧠 Regla rápida para clase

- Si quieres **estructura** → `innerHTML`
- Si quieres **solo texto** → `textContent`

4️⃣1️⃣ Casos de prueba para demostrar que lo has verificado

En una entrega, no basta con “que funcione”: interesa mostrar que lo has probado con casos representativos.

✅ Caso 1 — Una asignatura suspendida (no prácticas)

Notas: 6, 4.5, 7

- Suspensos = 1 → alerta: “asignatura pendiente, no prácticas”.
- Media = 5.83 → calificación: **APROBADO**.

✅ Caso 2 — Dos asignaturas suspendidas (repite)

Notas: 3, 4, 8

- Suspensos = 2 → alerta: “debes repetir el curso”.
- Media = 5.00 → calificación: **APROBADO** (ojo: la media puede engañar, pero la regla de suspensos manda).

📌 Este caso es clave porque demuestra que:

> no solo importa la media, también la condición “tres notas ≥ 5”.

✅ Caso 3 — Matrícula de honor

Notas: 10, 10, 10

- Suspensos = 0 → alerta: “puede iniciar prácticas”.
- Media = 10 → calificación: **MATRÍCULA DE HONOR**.

🔚 Conclusión integradora (para tus apuntes de Clase 6)

Esta práctica demuestra el objetivo final de la clase:

- HTML define **estructura**
- JavaScript controla **funcionalidad**
- El DOM se manipula en orden correcto:
  1. **DOMContentLoaded**
  2. Selección de nodos
  3. Eventos (`submit` , `click`)
  4. Sustitución de contenido (`innerHTML`)
  5. Ajuste de atributos (`setAttribute`) y estilos (`style`)

Resultado: una página que **se comporta como una mini-app**, sin depender de lógica en HTML.

3️⃣9️⃣ Tarea Obligatoria — Práctica de Nodos (DOM)

En este bloque se integra una **tarea obligatoria evaluable** cuyo objetivo es comprobar si el alumno ha interiorizado **cómo se construye el DOM desde JavaScript**, sin depender de HTML estructural.

La práctica se apoya directamente en los conceptos vistos en clase:

- jerarquía DOM (padre → hijo)
- creación dinámica de nodos
- inserción mediante `appendChild()`
- uso de `createTextNode()`
- optimización mediante bucle `for`

📌 **Restricción clave**:

> Toda la estructura debe crearse **exclusivamente desde JavaScript**.

Estructura solicitada por la profesora

La interfaz a construir debe responder a esta disposición lógica:

```
document
 └── table
     └── tr
         ├── td
         │   ├── img
         │   └── a
         ├── td
         │   ├── img
         │   └── a
         └── td
             ├── img
             └── a
```

Cada columna contiene:

- una **imagen**
- un **enlace**
- ambos dentro de una **celda de tabla**

📌 El elemento **contenedor principal** es la tabla (`table`), que debe añadirse al documento con `appendChild()` .

4️⃣0️⃣ Flujo mental aplicado (metodología de la clase)

Antes de escribir código, se sigue el flujo de trabajo insistido en clase:

1. Pensar qué nodos se necesitan
2. Definir quién será el **padre directo** de cada nodo
3. Crear los nodos con `createElement()`
4. Crear el texto con `createTextNode()`
5. Insertar los nodos con `appendChild()`
6. Repetir la estructura con un bucle `for`

📌 **Crear no es mostrar**:
el nodo no existe en el DOM hasta que se inserta en la jerarquía.

4️⃣1️⃣ Código completo de la práctica (versión optimizada con `for`)

📁 Estructura del proyecto

```
TareaObligatoria_Nodos/
├─ index.html
├─ style.css
└─ app.js
```

📄 `index.html`

```html
<!doctype html>
<html lang="es">
<head>
  <meta charset="utf-8">
  <title>Tarea Obligatoria Nodos</title>
  <link rel="stylesheet" href="style.css">
  <script src="app.js" defer></script>
</head>
<body>
  <!-- Toda la estructura se genera desde JavaScript -->
</body>
</html>
```

📄 `app.js`

```javascript
document.addEventListener("DOMContentLoaded", () => {

  const tabla = document.createElement("table");
  const fila = document.createElement("tr");

  for (let i = 1; i <= 3; i++) {
```

```
    const celda = crearCelda(i);
    fila.appendChild(celda);
  }

  tabla.appendChild(fila);
  document.body.appendChild(tabla);

});

function crearCelda(indice) {

  const td = document.createElement("td");

  const contImg = document.createElement("div");
  const img = document.createElement("img");
  img.setAttribute("src", `img/foto${indice}.jpg`);
  img.setAttribute("alt", `Imagen ${indice}`);

  img.addEventListener("error", () => {
    contImg.innerHTML = "";
    contImg.appendChild(
      document.createTextNode("IMAGEN")
    );
  });

  contImg.appendChild(img);

  const contLink = document.createElement("div");
  const enlace = document.createElement("a");
  enlace.setAttribute("href", "#");
  enlace.appendChild(
    document.createTextNode(`ENLACE ${indice}`)
  );

  contLink.appendChild(enlace);

  td.appendChild(contImg);
  td.appendChild(contLink);

  return td;
}
```

4️⃣2️⃣ Relación directa con los conceptos de la Clase 6

Esta práctica consolida los siguientes puntos teóricos:

- **Jerarquía DOM**
  Cada nodo se crea sabiendo quién es su padre.
- **`appendChild()` como paso crítico**
  Sin inserción no hay renderizado.
- **DOM dinámico**
  Los nodos se crean, sustituyen y modifican en tiempo real.

- **Uso correcto de bucles**
  El `for` evita código repetitivo y permite escalar la estructura.
- **Separación de responsabilidades**
  HTML carga el JS, JS construye la interfaz.

4️⃣3️⃣ Análisis en profundidad del código y de los métodos usados

Este apartado explica **qué hace cada método**, **por qué se usa aquí** y **qué parte de la jerarquía DOM afecta**, relacionándolo con la idea central de la clase:

> "El DOM es una estructura arbórea: si no defines padre → hijo y no insertas con `appendChild()`, el nodo no existe para el navegador."

🔹 `document.addEventListener("DOMContentLoaded", callback)`

En esta práctica el JavaScript **crea nodos** y **los inserta**. Eso significa que necesitamos ejecutar el código **cuando el DOM ya está listo**.

- `DOMContentLoaded` se dispara cuando el navegador ya ha construido la estructura del documento (árbol DOM).
- Es el "evento principal" que se remarca en clase para evitar errores como:
  - seleccionar elementos que aún no existen
  - insertar nodos antes de que exista `body`

En el código:

- se crea la tabla
- se crea la fila
- se construyen 3 celdas
- y al final se inserta todo en `document.body`

Todo eso debe ocurrir con el DOM listo.

🔹 `document.createElement(tagName)`

Es el método base para **crear nodos de tipo elemento**.

En la práctica se usa para crear:

- `table` (contenedor principal)
- `tr` (fila)
- `td` (celdas)
- `div` (contenedores internos para separar visualmente zona imagen / zona enlace)
- `img` (imagen)
- `a` (enlace)

Punto clave:

- `createElement()` crea el nodo **solo en memoria**
- todavía no pertenece al árbol DOM
- por tanto, no se ve en la página

Esto conecta con el concepto de “crear ≠ mostrar”.

🔹 `appendChild(hijo)` (la pieza crítica)

`appendChild()` es el método que **mete un nodo en la jerarquía DOM** como hijo del padre.

Se usa de forma encadenada para construir el árbol:

1. Celdas dentro de la fila
   - `fila.appendChild(celda)`
2. Fila dentro de la tabla
   - `tabla.appendChild(fila)`
3. Tabla dentro del documento
   - `document.body.appendChild(tabla)`

Dentro de cada celda también se aplica la misma lógica:

- el `img` se cuelga de `contImg`
- el `a` se cuelga de `contLink`
- ambos contenedores se cuelgan de `td`

📌 Idea central:

- si no se hace `appendChild()`, el nodo no aparece
- porque no forma parte del árbol (no tiene padre real)

🔹 `for (let i = 1; i <= 3; i++)`

El bucle es la parte “depurada” de la entrega.

Se utiliza porque la estructura es repetitiva:

- 3 columnas idénticas
- misma jerarquía interna
- solo cambia el índice (1, 2, 3)

En cada vuelta el bucle hace:

- crear una celda completa con `crearCelda(i)`
- colgarla en la fila

Esto demuestra una competencia clave:

- no solo saber crear nodos
- sino saber automatizar la creación de estructura DOM

🔹 `function crearCelda(indice)`

Esta función encapsula el patrón repetitivo: “una celda”.

Encapsularlo permite:

- limpiar el código principal (`DOMContentLoaded`)

- reutilizar la lógica
- mantener el mismo flujo mental:
  - crear → definir padre → insertar

La función devuelve el `td` ya construido, para que el bucle solo tenga que hacer:

- `fila.appendChild(celda)`

Esto es muy coherente con lo visto en funciones:

- una función “produce” un nodo preparado
- y se integra en la jerarquía donde corresponda

🔹 `setAttribute(nombre, valor)`

Este método permite añadir o modificar atributos HTML.

Se usa especialmente en imágenes:

- `src` para indicar el archivo
- `alt` para accesibilidad y semántica

Ejemplo del código:

- `img.setAttribute("src", \` img/foto${indice}.jpg `)`

Esto demuestra algo importante:

- el atributo se genera dinámicamente
- el nombre del archivo depende de `indice`
- eso hace que el DOM sea “programable”

🔹 `document.createTextNode(texto)`

Este método crea un **nodo de texto real**, no una string suelta.

Se usa cuando queremos que el enlace tenga contenido textual:

- `ENLACE 1`, `ENLACE 2`, `ENLACE 3`

En el código:

- se crea el nodo texto
- se cuelga del `<a>` con `appendChild()`

Esto conecta con lo que se comenta en clase:

> “Si creo texto, puedo crear un nodo de texto y añadírselo al elemento donde quiero que viva.”

🔹 `img.addEventListener("error", ...)` (sustitución / DOM dinámico)

Aquí aparece un concepto potente: **el DOM puede adaptarse**.

Si el archivo `img/fotoX.jpg` no existe, se dispara el evento `error`.

En ese momento el código hace:

- vaciar `contImg`

- insertar un texto “IMAGEN”

Esto no es estético: es didáctico.

Demuestra:

- que un nodo puede ser reemplazado
- que el DOM es dinámico
- que los eventos también aplican a elementos creados desde JS

🔹 `innerHTML = ""` (limpieza controlada)

Se usa solo como forma rápida de “vaciar” el contenedor de imagen cuando hay error.

En términos de DOM:

- elimina los hijos actuales del contenedor
- deja el nodo `contImg` listo para recibir un nuevo hijo (texto)

Después se inserta el texto con el método correcto:

- `appendChild(createTextNode(...))`

🧠 Lectura final del flujo completo

Si se traduce el código a “lenguaje humano” siguiendo la metodología de la profesora:

1. Espero a que el DOM esté listo (`DOMContentLoaded`)
2. Creo el contenedor principal (`table`)
3. Creo el nodo fila (`tr`)
4. Repito 3 veces:
   - creo una celda (`td`)
   - creo una imagen y un enlace (nodos hijos)
   - cuelgo esos nodos dentro de la celda
   - devuelvo la celda lista
5. Cuelgo las 3 celdas de la fila
6. Cuelgo la fila de la tabla
7. Cuelgo la tabla del documento (`body`)

📌 Resultado:

- no hay estructura en HTML
- la estructura nace en JS
- y queda integrada en el árbol DOM correctamente

4️⃣4️⃣Qué demuestra esta tarea (a nivel evaluable)

✔️ Comprensión real del DOM
✔️ Capacidad de construir estructura sin HTML
✔️ Uso correcto de nodos y texto
✔️ Aplicación de bucles en un contexto real
✔️ Pensamiento estructural (no mecánico)

📌 **Conclusión docente**:

> Si se domina esta práctica, el alumno está preparado para trabajar con DOM avanzado y frameworks.
